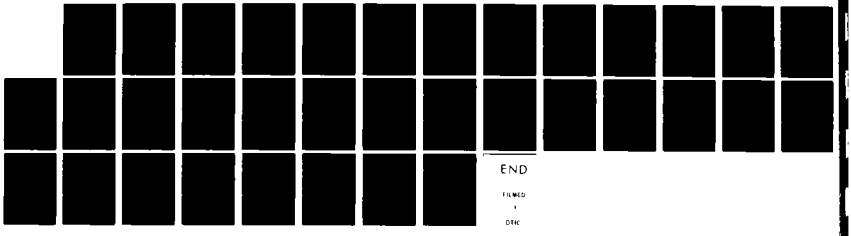
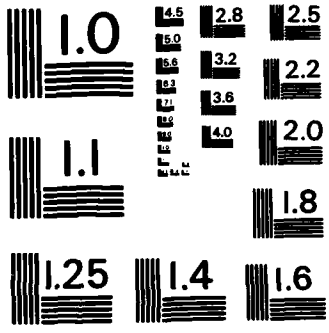
AD-A121 432 DEVELOPMENT OF A VOICE FUNNEL SYSTEM(U) BOLT BERANEK AND NEWMAN INC CAMBRIDGE MA R D RETTBERG OCT 82 BBN-5009 MDA903-78-C-0356 1/1

UNCLASSIFIED F/G 17/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS - 1963 - A

Bolt Beranek and Newman Inc.

(12)

Report No. 5009

Development of a Voice Funnel System

Quarterly Technical Report No. 15
1 February 1982—30 April 1982

October 1982

Prepared for:
Defense Advanced Research Projects Agency

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. A121492	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Development of a Voice Funnel System Quarterly Technical Report No. 15		5. TYPE OF REPORT & PERIOD COVERED Quarterly Technical 1 Feb - 30 April 1982 6. PERFORMING ORG. REPORT NUMBER 5009
7. AUTHOR(s) R. D. Rettberg		8. CONTRACT OR GRANT NUMBER(s) MDA903-78-C-0356
9. PERFORMING ORGANIZATION NAME AND ADDRESS Bolt Beranek and Newman Inc. 10 Moulton Street Cambridge, MA 02238		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA 1400 Wilson Boulevard Arlington, VA 22209		12. REPORT DATE October 1982 13. NUMBER OF PAGES 30
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution Unlimited

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Voice Funnel, Digitized Speech, Packet Switching, Butterfly Switch, Multiprocessor.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This Quarterly Technical Report covers work performed ~~during the period noted~~ on the development of a high-speed interface, called a Voice Funnel, between digitized speech streams and a packet-switching communications network.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Report No. 5009 Bolt Beranek and Newman Inc.

DEVELOPMENT OF A VOICE FUNNEL SYSTEM

QUARTERLY TECHNICAL REPORT NO. 15
1 February 1982 to 30 April 1982

October 1982

This research was sponsored by the
Defense Advanced Research Projects
Agency under ARPA Order No.: 3653
Contract No.: MDA903-78-C-0356
Monitored by DARPA/IPTO
Effective date of contract: 1 September 1978
Contract expiration date: 31 December 1982
Principal investigator: R. D. Rettberg

Prepared for:

Dr. Robert E. Kahn, Director
Defense Advanced Research Projects Agency
Information Processing Techniques Office
1400 Wilson Boulevard
Arlington, VA 22209

Table of Contents

1. Introduction

This Quarterly Technical Report, Number 15, describes aspects of our work performed under Contract No. 903-78-C-0356 during the period from 1 February 1982 to 30 April 1982. This is the fifteenth in a series of Quarterly Technical Reports on the design of a packet speech concentrator, the Voice Funnel.

In Quarterly Technical Report Number 5, we presented a pre-implementation description of the part of the operating system which is responsible for object management. In this report, we present a description of the Object Management System as it was finally implemented.

2. Object Management

The Object Management System unifies many data structures in the Butterfly operating system, Chrysalis. These data structures support the central mechanisms of the operating system such as free and allocated memory, processes, and events. The Object Management System gives us a simple 32-bit handle on these data structures and a simple way of referencing these objects. For example, a process is represented by a data structure called a Process Control Block; the Object System gives us a "Process Handle" that is used throughout the system to represent the process and that can be used to find the associated Process Control Block.

The Butterfly, like the Pluribus, is based on the notion that the ability to share memory is important if a multiprocessor is to achieve high performance. At the same time, our experience with the Pluribus indicates that if we are to have shared memory, we must provide adequate mechanisms to manage that memory. For this reason, the hardware of the Butterfly Multiprocessor implements a segmented address space with access protection on each segment.

One of the roles of the Object Management System is to help the programmer to manage this form of address space. One of the ways it does this is by letting the user define memory-based objects of many types and providing system calls which place them

into and remove them from segments in his address space in a natural way.

Since each memory object is represented by an Object Handle that is unique across the whole machine, a process can share a data structure with other processes by giving those processes a copy of the Object Handle. In this way, the notion of an Object Handle forms a basic mechanism for inter-process communications while it also helps the programmer deal with the segmented nature of the machine's address space in a comfortable manner.

The Object Management System as it is currently implemented provides only a limited degree of protection against programming errors and is not intended to protect against malicious users. The memory management hardware supports access controls for each segment. The exact forms of access that are permitted are described in Quarterly Technical Report Number 12. Only those objects that are mapped into the address space of a process can be modified by that process and the only modifications that are permitted are those specified by the hardware.

In a more comprehensive protection scheme, it would be necessary to control which objects may be mapped in by a process and what forms of access are legal. Right now, a process can ask to have access to any object and may request read or read and write access modes. This is adequate for the funnel and is very useful in detecting many software bugs.

2.1 Object Types

At this time, the object system supports the following 10 different types of objects; others will be added as they become necessary.

- Event Block.
- Timer Event Block.
- Process Control Block.
- Process Template.
- Channel Control Block.
- I/O device block.
- Variable length data block.
- System queue block.
- Dual Queue header.
- General user object.

The "Event Block" is the data structure for Events. Events are used to control process activation and for inter-process communications.

The "Timer Event Block" is used to wake a process at some specified time in the future.

The "Process Control Block" (PCB) is the primary data structure for a process. It includes address space information, scheduling information and saved process state. Each process is represented by its own PCB even if several processes are sharing the same code segment.

The "Process Template" contains a pointer to the code and initialized data of a process so that copies of a process can be created easily. The Process Template is much like the object

file for a program.

The "Channel Control Block" (CCB) is the data structure used by the I/O system to maintain chained buffers for input and output. Quarterly Technical Report Number 10 describes CCBs and the I/O system in detail. [The current structure for this object is incompatible with the data structure described in the previous section. We plan to fix it in a future I/O board microcode release.]

The "I/O Device block" represents one physical I/O device in the system.

The "Variable length data block" is not an object at all, but simply a way of reserving space in Segment F8 for special operating system data structures.

The "System Queue Block" is used by the operating system to maintain some of its internal queues and is used by I/O processes to maintain special queues of CCBs as well.

The "Dual Queue" is a microcoded synchronization and task allocation mechanism.

The "General User Object" (also called a "memory object") has both an object header and a piece of the Processor Node's physical memory. The object header is located in an area of memory used by the system for object headers; the memory portion is allocated from the free physical memory of the Processor Node.

The user creates this type of object in the process of memory allocation. The memory is available in the sizes supported by the memory management hardware (see QTR 12). With the handle which represents this physical memory, any process in the machine may map the memory into its address space.

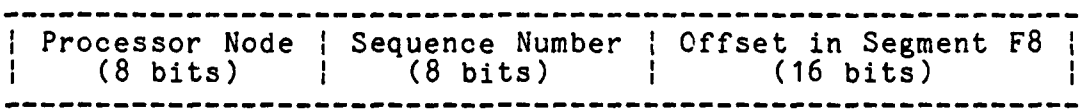
Because many processes may have access to a memory object at the same time, it may be difficult for the application software to determine when the memory object is no longer in use and should be deleted. This problem is further complicated by the possibility that a process may terminate normally or abnormally while it has the object mapped into its address space.

To solve these problems, the Object Management System controls the allocation of this type of object. The object header contains a usage count to keep track of how many processes have it mapped in. Whenever a process maps it in, the use count is incremented. Whenever it is unmapped, the use count is decremented. For a memory object to be deleted by the operating system, two conditions must be met. First, the use count must be zero. Second, the object must have been explicitly deleted. This lets a process hand a memory object to another process without concern for whether the receiver has mapped the object in before the sender unmaps it. It also lets a process delete a memory object without having to track down outstanding pointers to that object.

2.2 Object Handles

An Object Handle is like a pointer to the data structure that is associated with an object. However, unlike a simple pointer, an Object Handle must be interpreted before it can be used. In return for this additional processing overhead, an Object Handle is much safer to use in a large system than a simple pointer would be.

In practice, the structure of an Object Handle is not important to application programs -- the handle is passed as a single unit to the system routines that implement the object system and it is stored as a 32-bit "long" in the machine. Internally, the system breaks an Object Handle into three fields as shown below.



The first field specifies which Processor Node contains the header data structure for this object.

The sequence number field is for debugging and error detection purposes. It is compared with a corresponding sequence number field in the object's data structure whenever the handle is used. In this way, we detect obsolete Object Handles and are also often able to detect when a "handle" is not an Object Handle at all.

The reader will notice that because the sequence number field is only 8 bits, once the same data structure has been reused 256 times, obsolete handles are no longer detected. In other words, Object Handles are not totally unique. This means that in operational software, the programmer should not count on Object Handles to detect when resources have become unavailable. Instead, he should explicitly clean up after failures. However, as a debugging aid 8-bits should be adequate since it is very unlikely that an object data structure will have been reused exactly 256 times at the time that an obsolete handle is used.

The final portion of the handle, the "Offset in Segment F8", is actually a pointer. In order to understand its function, we must review the memory layout of the machine. The 24-bit address of the 68000 is divided into an 8-bit segment number and a 16-bit offest. There are 256 segments in a process, numbered from 0 to hex FF. Segment zero is used for hardware support. Segments F8 through FF are used for system functions.

By convention, we assign segment F8 to operating system data structures and provide a memory allocation package that treats this segment by itself. When we allocate objects, we always do it from segment F8. We adopt the further convention that physical memory starting at location zero is mapped into segment F8 in every Processor Node. Thus, the data structure corresponding to an Object Handle is found on the Processor Node specified in the handle, and begins in physical memory at the

offset specified in the handle. These conventions permit the microcode of the Processor Node Controller to find the data structures of the object system so that the microcode can provide high-level functions for the operating system.

This strategy makes good use of the bits in the handle and is easy to interpret but limits the number of objects that are possible on a node to what fits in a single segment. Even with this limit a very large number of objects can be supported. If we find it necessary to change this strategy in the future, the impact on the application software should be minimal since the application software considers the Object Handle to be atomic.

2.3 Object Data Structures

The data structures for all objects have a common form. The first entry is a pointer to another object of the same type. This is followed by the structure given below. At the end is a structure that depends on the type of the object.

```
{
    char unsigned o_type;      /* block type        */
    char bits     o_flags;     /* flags, see below */
    char unsigned o_stype;     /* block subtype     */
    char unsigned o_seqno;     /* sequence number   */
    bits          o_prot;      /* protection bits   */
    OID           o_owner;     /* owners Object Handle */
}

/* definitions for o_flags */

    O_FROZEN      0x08         /* object is frozen  */
    O_HAS_NAME    0x10         /* object has a name */
    O_FREE        0x20         /* object is not in use*/
    O_DELETED     0x40         /* object has been deleted */
```

Figure 1 - Uniform Object Header

"o_type" specifies the type of this object. "o_stype" permits the existence of objects that act in the same way from the point of view of the object system, but are to be handled differently by the user programs. "o_seqno" is the sequence number that corresponds to the sequence number in the handle. "o_prot" is currently unused.

The flags in the object header control special states of the object. When a system call or microcode function involving an object returns an error, the flags are often reported. Some of the objects can be "frozen" so that their normal functions are prohibited. In this state, they can be verified for correctness, debugged, or prepared for deletion.

The flag O_HAS_NAME is set whenever the object has a corresponding entry in the system-wide name table. The name table provides a service that couples Object Handles (or in fact

any long value) to strings that name the object. The table is arranged by type so that the same name can be used by different types. All of the object types are filed in this system under the type 256.

By using the name table, a process can find the important system-wide resources that it needs. For example, events which signal important occurrences will have names. Many processes will also have names. As a debugging aid, a command is available which types out the names of objects.

The two final flags are used by the object allocation routines to manage the allocation and deletion of objects.

The "o_owner" field contains the Object Handle of the object that owns it. Often, this field will contain the handle for the process that created the object. The owner can change this field by calling a function which "disowns" the object and passes ownership to another object. If the owner of an object is zero, then the system is considered to own the object. Sometimes it is useful to have a data object own other data objects. For example, it may be useful to have a Free Queue own its buffers. In fact, we use ownership to ensure that deletion occurs in the proper order (e.g. that the Free Queue gets deleted before the Buffer Pool does) and that deletion of one component of a Buffer Pool (the Free Queue) guarantees deletion of all data structures that depend on it.

The notion of ownership is normally used to figure out when to delete an object when a process terminates. For most objects, when a process is deleted, the system arranges to delete its resources directly, but in some cases, it is easier for the system to simply mark the process deleted and let the garbage collector delete it later. The operating system contains a garbage collection process which periodically examines each object on the Processor Node. If the owner is no longer valid, then the object is deleted.

2.4 Object Management Functions

On the following pages, we have provided the documentation for the Chrysalis system calls which manipulate objects for the user or for other operating system routines. While this documentation reflects that current implementation, Chrysalis is constantly evolving and these routines are subject to change without notice.

In addition to these routines, there are routines which are used to manipulate specific types of objects such as events or Dual Queues. These routines will be described in the Chrysalis operating system manual.

Title: **Make_Obj**

Function: Creates a memory-type object.

Arguments:

1. char Subtype of the object
2. int Processor node on which to create object (-1 => local)
3. int Size of the memory of this object in bytes
4. bits Desired protection code, (0 => use default)

Return Value: OID -- Object Handle for the new object.

Possible Exceptions:

CONSISTENCY	Specified processor is not up
NOMEM	F8 is out of memory
NOMEM	Memory is not available

Files: /usr/butterfly/chrys/prot/cou.c68

Description:

Gets an object of the type which represents a block of user memory on the specified Processor Node. Minus one specifies that the current processor node is to be used. The sub_type field is available to the application program for its own purposes. The memory block allocated is large enough to provide the size block requested, and is one of the following sizes: 0, 256 bytes, 512, 768, 1024 (1K bytes), 1536, 2K, 3K, 4K, 6K, 8K, 12K, 16K, 24K, 32K, 64K. Notice that 48K is omitted from this sequence and that 64K bytes is the largest size available.

All allocatable memory is cleared during Chrysalis initialization, and the data you store into a memory block will be cleared automatically when the object is finally deleted. Therefore you can count on Make_Obj returning a memory block which has been set to all zeros.

Bugs:

Title: Del_Obj

Function: Delete an Object and recover resources.

Arguments:

1. OID Object Handle of the object to delete

Return Value: None.

Possible Exceptions:

BADHANDLE Invalid Object Handle

Files: /usr/butterfly/chrys/obj/cou.c68

Description:

Deletes the object corresponding to the OID supplied. It also deletes any name associated with the object, and frees up and clears memory owned by the object.

For memory-type objects, the delete does not take effect until the object has been Unmapped by all the processes which were using it. This type of object includes a 'DELETED' bit, and a use counter which is incremented and decremented by Map_Obj and Unmap_Obj respectively. Unmap automatically calls Del_Obj if the use counter goes to zero for a 'DELETED' object.

Del_Obj understands how to do the special processing needed to delete all types of object, including processes and process templates. If you delete a dual queue for which processes are waiting, the associated event blocks are automatically posted with a data value of NULL.

The garbage collector calls Del_Obj whenever it encounters an object whose owner field contains an invalid Object Handle. Although it is good practice to explicitly delete objects you own before exiting, the garbage collector will generally succeed in cleaning up after processes which terminate unexpectedly.

Bugs: None.

Title: **Map_Obj**

Function: Maps the memory of a memory-type object into the user's address space.

Arguments:

1. OID Object Handle for the desired object
2. int Desired segment number (0 => use highest free segment)
3. bits Desired access mode

Return Value: char* -- Pointer to the memory region.

Possible Exceptions:

BADHANDLE	Invalid Object Handle
CONSISTENCY	Object has no memory
CONSISTENCY	Invalid segment specified
NOMEM	Process has no free segments
CONSISTENCY	Requested access is not allowed

Files: /usr/butterfly/chrys/prot/cou.c68
/usr/butterfly/h/public.h

Description:

This routine maps in the memory of a memory-type object. It accepts an Object Handle and a segment number. If segment number zero is specified an unused segment is allocated and added to the process's address space. If a specific segment is specified it must be free. The memory portion of the object will fill the segment exactly.

If this routine is called in user-mode, only those modes allowed by the protection bits in the OAB are legal. There are eight hardware protection modes, as listed in the following table.

R_____	Only kernel mode routines can read this segment.
RW____	Kernel mode routines can read or write it.
R__r__	Kernel or user-mode routines can read it.
RW_r__	Kernel read/write, user read only.
RW_rw_	Read/write in either mode.
R_Xr_x	Read or execute code in either mode.
R__r_x	Read in either mode, or execute user-mode code.
RWXrwx	Anything goes.

Bugs: Currently, the protection bits in the OAB are ignored, and user-mode programs may request one of the following three modes: RW_rw_, RW_r__, and R__r__.

Title: Unmap_Obj

Function: Removes an object from the address space of a process.

Arguments:

1. OID Object Handle for object to unmap
2. char* Pointer to the object

Return Value: None.

Possible Exceptions:

BADHANDLE Invalid Object Handle
CONSISTENCY Invalid object address
CONSISTENCY Object/address mismatch

Files: /usr/butterfly/chrys/prot/cou.c68

Description:

This is the inverse of the Map_Obj call. It removes the object from the process's address space and marks the segment unused. If the object's use count goes to zero and its 'DELETED' bit is set, Del_Obj is called to recover the object's resources.

Bugs: None.

Title: map_oab

Function: Maps in the Object Attribute Block of an object.

Arguments:

1. OID Object Handle for desired OAB
2. int Object type expected (0 => any type)
3. int Desired segment number

Return Value: OAB* -- Pointer to the OAB.

Possible Exceptions:

BADHANDLE Invalid Object Handle
CONSISTENCY Invalid segment specified

Files: /usr/butterfly/chrys/prot/cou.c68

Description:

Map the OAB for the specified object into the process's address space. The argument "type" is the desired type of the object. A type of zero will cause this routine to accept any type of OAB. If the object is on this processor node, then segment F8 will be used. Otherwise if segment zero is specified, the highest numbered free segment in the user's address space will be used; else if a free user segment is specified, it is used.

OABs should be unmapped using unmap_oab.

Some Chrysalis routines must use segments outside the user's SAR group. To provide for this, if a kernel segment (above F8) is specified, it is used but only if map_oab was called with interrupts inhibited. Inhibiting interrupts 'locks' the use of these segments, and they must remain inhibited until unmap_oab has been called.

In some cases where Chrysalis needs to map an OAB and check that it is local, we specify segment F8 (without inhibiting interrupts). This will fail if F8 was not already set correctly, i.e., if the OAB is remote. In this case it is not necessary (but is permissible) to call unmap_oab.

Bugs: Any process can map in any OAB, with RW_r__ access.

Title: **map_my_oab**

Function: Maps in a local OAB owned by this process.

Arguments:

1. OID Object Handle for desired OAB
2. int Object type expected (0 => any type)

Return Value: OAB* -- Pointer to the OAB.

Possible Exceptions:

BADHANDLE Invalid Object Handle

Files: /usr/butterfly/chrys/prot/cou.c68

Description:

Like map_oab except that only local objects owned by this process are accepted. A pointer to the specified OAB is constructed using segment F8. This routine is used to enforce the restriction that an object be owned by the current process on the current node (for example, when manipulating event blocks).

The argument "type" is the desired type of the object. A type of zero will cause this routine to accept any type of OAB.

Since F8 is always used, it is unnecessary (but permissible) to call unmap_oab when you finish using the returned pointer.

Bugs:

Title: **pnc_widen macro**

Function: Maps in a local OAB with no checking.

Arguments:

1. OID Object Handle for desired OAB

Return Value: OAB* -- Pointer to the OAB.

Possible Exceptions: None.

Files: /usr/butterfly/h/pnc.h

Description:

Like map_oab except that only local objects can be referenced, and no checking is done. A pointer to the specified OAB is constructed using segment F8. This macro is used when you are sure an Object Handle is local and is valid, and want to map in its OAB as quickly as possible.

Since F8 is always used, it is unnecessary (but permissible) to call unmap_oab when you finish using the returned pointer.

Bugs:

Title: unmap_oab

Function: Removes an OAB from an address space.

Arguments:

1. OAB* Pointer to the OAB

Return Value: None.

Possible Exceptions:

CONSISTENCY Invalid oab pointer

Files: /usr/butterfly/chrys/prot/cou.c68

Description:

Removes the OAB from the user's address space and marks the SAR free. This routine is a nop for local objects mapped via segment F8.

If a kernel segment (above F8) is specified, interrupts must be inhibited.

Bugs: Not much checking is done. If the segment number is valid and in use, the segment is made free. Be careful not to use unmap_oab instead of Unmap_Obj, as that will invalidate the object's use count leaving it undeletable.

Title: Obj_OK

Function: Checks the validity of an Object Handle.

Arguments:

1. OID Object Handle to check

Return Value: int -- TRUE if ok, else FALSE.

Possible Exceptions:

CHECK Bus error -- Object Handle was garbage
CHECK Address error -- Object Handle was garbage

Files: /usr/butterfly/chrys/prot/cou.c68

Description:

This routine checks Object Handles for currency; its operation is undefined if its argument was never a process handle. However, NULL Object Handles and handles which specify invalid or non-running Processor Nodes will always return FALSE.

Obj_OK maps in the given object and checks its sequence number for validity. Bus/address errors are possible only if the alleged Object Handle was never valid or the switch fails.

Bugs: None.

Title: Disown_Obj

Function: Gives up ownership of an object.

Arguments:

1. OID Object Handle for the desired object
2. OID Object Handle for the new owner (or NULL)

Return Value: None.

Possible Exceptions:

BADHANDLE Invalid Object Handle
NOMEM Process has no free segments
CONSISTENCY Not your object
CONSISTENCY Can't disown events

Files: /usr/butterfly/chrys/prot/cou.c68

Description:

Gives up the ownership of an object. If the new owner argument is NULL, ownership passes to the system, and the object remains allocated until explicitly deleted. Otherwise the new owner must be a valid Object Handle (frequently a process handle), which inherits ownership of the object.

When an owner object is deleted, objects which it still owns are automatically deleted by the garbage collector. Of course memory-type deleted objects do not actually disappear as long as their memory is mapped in by any process.

Disowning event blocks is not allowed.

Bugs: The new owner argument is not checked; if it turns out to be invalid the object will be deleted by the garbage collector. Also, it is currently impossible to disown CCBs.

Title: **Copy_Obj**

Function: Creates a new copy of a memory-type object.

Arguments:

1. OID Object Handle of the old object
2. int Protection bits for the new object (0 => no change)
3. int TRUE to splice new object into a chain, else FALSE

Return Value: OID -- Object Handle of the new object.

Possible Exceptions:

BADHANDLE	Invalid Object Handle
NOMEM	Process has no free segments
NOMEM	F8 is out of memory
NOMEM	Memory is not available

Files: /usr/butterfly/chrys/prot/cou.c68

Description:

This routine creates a new copy, on this processor node, of the memory-type object specified. The OID for the new object is returned. If the specified protection is 0, the protection field is copied from the old object. If the chain argument is TRUE, the new object is spliced into the old object's o_link chain, and the o_owner field is copied from the old object. The subtype field is copied from the old object.

This routine will copy linked objects. A linked object is a group of objects, on the same processor node, linked together via the o_link field in the object header, and referenced by the Object Handle of the first object in the list.

This routine needs two free SARs in the current address space.

Bugs:

Title: Get_Obj

Function: Underlying object creation routine.

Arguments:

1. char Type code of object
2. char Subtype of the object
3. int Processor node on which to create object
4. int Size of the memory of this object in bytes
5. bits Desired protection code, (0 => use default)

Return Value: OID -- oid of object created.

Possible Exceptions:

CONSISTENCY Type is out of range
CONSISTENCY Specified processor is not up
NOMEM F8 is out of memory
NOMEM Memory is not available

Files: /usr/butterfly/chrys/prot/cou.c68

Description:

Gets an object of the specified type and subtype on the specified Processor Node. If the type is BT_OAB, then memory can be associated with the object and its length is as specified.

This routine must be called in kernel mode.

Bugs: Protection is not yet implemented.

Title: **Find_Value**

Function: Finds a value, given a name and type.

Arguments:

1. char* Asciz name
2. char Type of entry

Return Value: bits -- Value associated with the given name and type.

Possible Exceptions:

FAILED Name table entry was not found

Files: /usr/butterfly/chrys/prot/name.c68

Description:

Searches the global name table for the specified name and type. If the name is not found, an exception is thrown.

Bugs: None.

Title: Find_Name

Function: Finds a name, given a value and type.

Arguments:

1. char* Address at which to store asciz name
2. bits Value of entry
3. char Type of entry

Return Value: None -- name is copied to the specified address.

Possible Exceptions:

FAILED Name table entry was not found

Files: /usr/butterfly/chrys/prot/name.c68

Description:

Searches the global name table for the specified value and type. If the value is not found, an exception is thrown.

Bugs: None.

Title: **Name_Bind**

Function: Adds a name/value/type triple to the global name table.

Arguments:

1. char* Address of asciz name
2. bits Value of entry
3. char Type of entry

Return Value: None.

Possible Exceptions:

FAILED	Duplicate name in use
FAILED	Duplicate value in use
FAILED	Global name table is full
CONSISTENCY	Value is not an Object Handle

Files: /usr/butterfly/chrys/prot/name.c68

Description:

This routine creates a system-wide binding between a triple of an ascii name, a value, and a type. It can be used to find public objects, such as dual queues, and public parameters of any sort.

Low numbered types can be used as needed by application programs. Within a given type, both name and values must be unique; that is, a given name may appear only once under a given type, and the same goes for values.

High numbered types may have meaning to Chrysalis. So far the only type assigned is NTYPE_OBJ (0xff). This type can be used only to assign a name to an object. The name will be automatically deleted by Chrysalis if the object is deleted. The value assigned must be a valid Object Handle for a previously unnamed object, and the name must not conflict with other names of this type.

Bugs: None.

Title: **Name_Unbind**

Function: Removes a entry from the global name table.

Arguments:

1. bits Value of entry
2. char Type of entry

Return Value: None.

Possible Exceptions:

FAILED Name table entry was not found

Files: /usr/butterfly/chrys/prot/name.c68

Description:

Searches the global name table for the specified value and type and removes it. If the value is not found, an exception is thrown.

Bugs: None.

DISTRIBUTION OF THIS REPORT

Defense Advanced Research Projects Agency
Dr. Robert E. Kahn (2)
Dr. Vinton Cerf (1)

Defense Supply Service -- Washington
Jane D. Hensley (1)

Defense Documentation Center (12)

USC/ISI
Danny Cohen
Steve Casner

MIT/Lincoln Labs
Dr. Clifford J. Weinstein (3)

SRI International
Earl Craighill (1)

Rome Air Development Center
Neil Marples - RBES (1)
Julian Gitlin - DCLD (1)

Bolt Beranek and Newman Inc.
Library
Library, Canoga Park Office (2)
S. Blumenthal
R. Bressler
R. Brooks
P. Carvey
P. Castleman
W. Edmond
G. Falk
J. Goodhue
S. Groff
E. Harriman
F. Heart
M. Hoffman
M. Kraley
A. Lake
W. Mann
W. Milliken
M. Nodine
R. Rettberg
P. Santos
G. Simpson
E. Starr
E. Wolf